Automatic Support for Requirements Validation

Assioua Yasmine, Rabéa Ameur-Boulifa, Patricia Guitton-Ouhamou, Renaud Pacalet

Automatic Support for Requirements Validation

Assioua Yasmine[†], Rabéa Ameur-Boulifa[*], Patricia Guitton-Ouhamou[†], Renaud Pacalet[*]
[*]LTCI, Télécom Paris, Institut Polytechnique de Paris, France
firstname.lastname@telecom-paris.fr
[†]Renault Software Labs, France
firstname.lastname@renault.com

***Abstract*—The automotive industry is currently going through rapid changes from a mechanical industry to one driven by innovation in electronics and embedded software. This significant change creates also significant challenges to the industry. One of the most important is the ability to create safe vehicles, emphasizing the importance of safety by design. This paper is intended to contribute to current activities working towards an industry-wide development of reliable and secure systems. Correct by design methodology, including formal methods, have the potential to improve dependability of systems in this domain. And their use at an early stage of the development process ensures faster time to market. In this paper, we present tool support for our approach that aims at integrating the formal analysis and verification of functional requirements from early stages of the development life cycle, by using model checking technique. From informal requirement specifications the tool delivers models. They will be used to produce evidences that the requirement specifications are realizable, otherwise it can guide their revision. The approach is illustrated by a case study based on a specific function of autonomous vehicles.**

***Keywords*—Requirements analysis, Reliable systems, Model-based design, Systems engineering.**

I. Introduction

Designing complex systems is a difficult task. Conventional development approaches provide a unified process for system development, from requirements analysis to implementation [14]. Such approaches play a major role in software engineering practices. But quality of designed products in terms of correctness and robustness still remains a hot spot. The issue of building a practical but accurate methodology for designing safe and correct systems still remains unsolved. Such approaches are generally based on late-stage validation relying on testing to check that the requirements specifications are correct or to detect possible flaws, which leads to hight-cost corrective measures or even huge financial losses. According to [6] and [22] financial losses caused by failures represent more than 5% of the overall turnover of the companies. On the other hand, the study published in [12] reports that 64% of all errors are introduced during requirements specification and design, and 36% of the errors are introduced during the implementation phase.

Test coverage techniques which are commonly used to ensure the quality of developed systems cannot be used at early stages. In contrast, formal analysis grants much higher potential to discover flaws during the design phase of a system. Yet, formal techniques for verification and validation often require a strong technical background that limits their usage especially in the industrial context.

In [32] we proposed a model-based approach for early validation of requirements that relies on formal methods. To facilitate the design of automotive software from requirements, we suggest enhancing the system design process with the use of formal methods, and to offer a tool that system designers can use to assess through a rigorous and systematic process that the developed systems is compliant with the predefined requirements. This paper is an extension of this previous work. In this one, we have significantly extended and implemented the proposed method. First, we enrich and improve the requirements expression language to address more applications in the automotive domain. The work resulting from this research is an approach to analyse adaptable and extensible templates that can be used to specify requirements in the automotive domain. Second, we introduce the implementation of the resulting template in Xtext an eclispe-based tool [7]. Finally, we significantly extend the empirical study by evaluating our approach with an additional use-case. We demonstrate the applicability of our tool on an industrial context through a realistic use-case: the Automatic Park Assist system.

While this work is conducted in a context of analysis of the software requirements embedded in vehicles, we believe that our approach is not related to any specific system. This is why throughout the paper we use the term generic "system" regardless of which system (software or physical components) it is referred to.

The rest of the paper is organised as follows: section II introduces the overall process for the formalization of requirements by showing the steps from input data to final output. Section III gives the structure of the requirements language used to specify automotive requirements. It resembles EARS requirements structure. Section IV provides the technical approach for the formalization of requirements. This includes the natural-like languages template for specifying requirements, as well as the association between templates and their semantics in the UPPAAL formalism, for the derivation of formal models. Section V validates our approach. Section VI surveys related work before concluding in section VII.

II. Approach for requirements validation

The approach we advocate for formally analysing and validating requirements is described in Fig. 1. Our tool, which

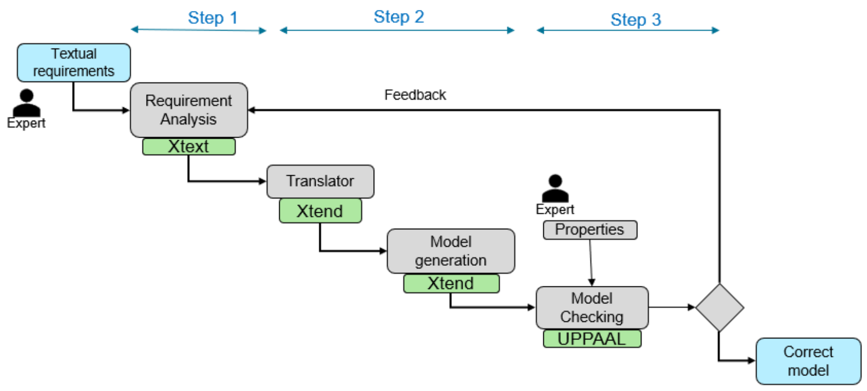


Fig. 1: An overview of the end-to-end approach for analysing formally automotive requirements

is a proof of concept, is built on the basis of open-source tools. It consists of three main parts:

- Step 1: To have a clear understanding of the automotive requirements, we conducted a deep analysis of textual requirements specifying different use-cases in our company. This work led to the identification of patterns and the establishment of their classification. From those patterns we created a grammar that gathers all the possible forms that a requirement can take. The classification is built regarding their structures and their role in the model construction. As with a programming language, the overall structure involves fixed terms (keywords), such as **while** and **shall**, that can be combined with free-form elements with no predefined scheme. The transformation of these structures into a formal notations allows to verify their completeness, consistency, and correctness by using automatic tools. The transformation task is implemented using the open-source software framework Xtext used for developing programming languages and domain-specific languages.
- Step 2: During this step we translate the requirements from their textual form to a model. We chose the UPPAAL automaton formalism as the modelling language for its ability to represent all the aspects desired and targeted by the analysis. In particular, the UPPAAL formalism supports various constructors, which gives it great power of expression. For instance, the use of expressions (for expressing guards and assignments) built over variables and parameters, and the use of synchronisation over channels. It also allows to model a global system by composition of subsystems (processes). The translation procedure is implemented using the open-souce Eclipse Xtend framework, also widely used for developing domain-specific languages.
- Step 3: The outcome of the construction of step 2 is either a valid model, or a non-valid model. In the former case, the automaton has an initial state, from the initial state there is a path to all other states, and the automaton is deterministic. A valid result provides an early evidence of requirement's consistency and correctness, it can then be used to a posteriori analysis and verification of properties. In the latter case, the malformations are shown to engineers who will correct or complete the requirements in an iterative process. To perform the analysis and verification task, we use the UPPAAL model checker: this tool allows to check automatically the consistency of the model against properties that can be expressed using specific languages (such as CTL language) or observer automata. In cases where properties are violated, the tool is able to provide a precise and useful feedback to the developer (engineer) to understand the source of the violation, and possibly how to fix it.

III. Requirements Specification

Many large firms, such as Renault, write technical specifications for the system under design or for the software application before getting started. The features and behaviour of the system or the application are described in a set of documents. It includes a variety of texts and graphics that defines the intended functionality required by the customer. At Renault, this document is called STRComp (from System Technical Requirement Component). This document includes textual templates that are requirements written in a constrained natural language, i.e, natural-like language with restricted syntax.

The analysis of the STRComp of different case studies studied during our work allowed us to classify the requirements into categories according to their role and their nature. Overall, we have identified three categories:

- *Interface requirements* specification defines interface for the system under design, in that it describes how to access the functionality provided by the system via variables or signals. This type of requirement gives the names of the variables and signals, and their domain. For example, the requirement scheme that is used to specify the signals sent by the system to the environment (subsystems) is of the following form:

```
<system> shall send from <actor> the signal <name>
[with the following values : (- <value>)+ ]
```

In the same way, requirement schemes are defined specifying the signals received by the systems:

```
<system> shall receive from <actor> the signal <name>
[ with the following values : - <value>)+ ]
```

- Functional requirements or *specification points* describe the desired behaviour of the system: what the system is intended to do and what conditions it must meet. The requirements of this category are written in a format close to the Easy Approach to Requirements Syntax (EARS) notation [24]. The structure of a specification point is made up of one or more patterns combined in the same order. A pattern is a compact and structured template; it consists of attributes and fixed syntax elements (keywords). In all the syntactical forms given below, the terms in bold are fixed syntax elements, while those between rafters are attributes. The generic pattern syntax used in our study is the following:
  1) A state-driven requirement defines the states of the system and the condition or triggering event that enables/disables actions:

     ```
     while <state> and <condition> [when <trigger>]
     <system> shall <action>
     ```

     An action describes the behaviour that the system should achieve. It is defined by a change of state or by an occurrence or consecutive occurrences of signal setting actions. It has the following format:

     ```
     <action> ::= switch to <state>
                | set <name> to <value>
                | activate <function>
                | release <function> control
     ```

     specifying a change of state, an update of a variable to a given value, an activation of a function, or a release of a control function of an actuator or sensor. The terms **activate** and **release** are introduced for readability reasons; they are used to raise requests that refer to an updating of signals, i.e. actions.
  2) An event-driven requirement defines how the system should behave in response to an effect (nominal or failure) of an action or an external stimulus that occurs:

     ```
     when <trigger>, <system> shall switch to <state>
     ```

  3) An action-driven requirement defines the action that is invoked when entering a certain state:

     ```
     when entering <state>, <system> shall (-<action>)+
     ```

  4) Some requirements define the conditions that enable a certain event to be issued. These conditions are complex and timed. A typical example of such requirements is the following:

     ```
     <system> shall detect <trigger>,
     if <name> = <value> for more than <delay>
     ```

     It specifies the conditions under which a certain event is triggered.

  where `<system>` a name of the system, `<name>` a name of signal, `<state>` a name of state, `<condition>` a condition that enables/disables actions, `<trigger>` an affect of an internal action or an external stimulus, and `<action>` a processing step, e.g. operations updating variables. These basic forms of requirements can be combined to specify complex requirements. For instance, the maximal representation of a state-driven requirement is given in Fig. 2. The identification of each pattern and its semantics allows to build the global model of the specified system.
- *Constraints* are also functional requirements that impose restrictions on the realisation of the system. They describe what constraints the realisation must satisfy to prevent various risky behaviour. They are of two types: those called *plausibility* that define rules about execution which are plausible and which are not, and those called *priorities* that define the priorities between its subsystems. As an example of a constraint:

  ```
  if <system> is in <state> and entrance conditions to
  <state> are satisfied, <system> shall switch to <state>
  ```

  this additional rule specifies the execution to be set aside and the execution to be imposed instead.

Although it is well-structured, this language, like RELAX [31] and Stimulus [20] languages, is classified as a natural language. It does not meet several assessing criteria for requirements engineering approaches, which are required by industry standards (as mentioned in ISO/IEC/IEEE 29148-2011 [19]). Among these criteria we find verifiability, which assesses whether an approach supports the possibility of formally verifying the properties of the requirements. This is inherently not the case for this language, as it is not tied to a formal semantics.

IV. From textual description to model

The main objective of our approach is to provide early evidence that a given set of requirement specifications are realizable. This objective is specifically related to the requirements formalisation challenge [26], [29]. The formalisation task refers to the transformation of requirements into formal models. In this section, we outline how formal models are derived from textual requirements.

For this purpose, we specified the requirement specifications using the state machine model. The main criteria used in selecting this formalism includes its precise formal semantics, but also its integration with automatic verification tools such as model-checkers. We have used the UPPAAL model checker [23] as it has proven to be successful and practical in various domains. This tool offers an integrated environment for analysing real-time systems based on networks of timed automata. It provides an editor, a symbolic simulator and a verifier, for modelling, enabling examination of dynamic executions, and verifying (by covering exhaustive dynamic behaviour).

A. UPPAAL *Model*

The model-checker UPPAAL is based on the theory of timed automata. Within this tool, a system is modelled as

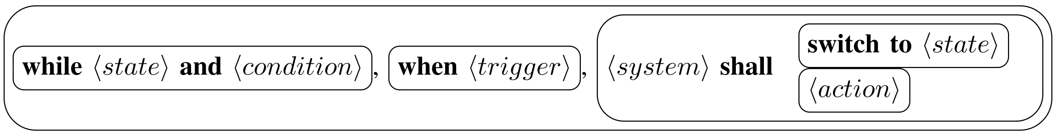


Fig. 2: The generic syntax of state-driven requirements

a network of timed automata that communicate in a synchronous fashion using so-called channels. A timed automaton is a classical finite-state machine extended with clocks. Each transition (edge) of such an automaton can be decorated with three (optional) labels:

$$\mathbf{S} \xrightarrow[update]{guard \quad sync} \mathbf{S}'$$

- *guard* expressing a condition on the values of the variables, which must be satisfied for the transition to be fired.
- *sync*, to represent synchronisation. Automata can synchronise over channels. The synchronisation mechanism in UPPAAL is a hand-shaking synchronisation: two processes take a transition at the same time on a common channel **e**, one will have a transition labelled **e!** to identify the sender and the other a transition labelled **e?** to identify receivers. UPPAAL offers urgent channels to force a synchronisation as soon as it is possible. It also supports the notion of broadcast channels that allow 1-to-many synchronisations.
- *update* a set of actions, which are expressions with a side-effect, i.e, assignment of variables or reset of clock. They may also be functions calls.

Note that S and S' are called *locations* in UPPAAL. A state of a UPPAAL model is defined by the locations of all automata being part of the model, the clock values, and the values of the variables. This other feature of UPPAAL is very useful for the applicability of our approach in an industrial context. It leads to a reduction of the state-space representation: automata with an infinite number of states can be represented by a finite set of symbolic states.

In addition to the network of automata, UPPAAL model includes a declaration part, which contains declarations of clocks, (global and local) variables, synchronisation channels, and constants manipulated by automata.

B. Model Construction

We provide an automatic and a systematic stepwise approach for transforming specification requirements into UPPAAL models. We first start with a preprocessing phase for unifying grammatical notations, e.g. unification of the letter case of attributes, and for defining a basis for modelling. Once the preprocessing is complete, all the requirements are translated systematically into automata that can model them. At the end of the translation we obtain an UPPAAL model, a network of (timed) automata, representing all the requirements.

a) Declaration part: Given a set of interface requirements, we get a set of variables. We translate each signal into a variable with the same name and definition domain. This part will be completed by the declaration of the channels as they are being created.

b) Automata generation: They are incrementally built by addressing all functional requirements (specification points and constraints). To do this, the translation procedure relies on an interpretation function denoted $[\![.]\!]$ that translates each textual item to a corresponding UPPAAL item. To build the global model we provide a systematic stepwise procedure:

1) During the first phase, the main automaton is derived from the state-driven requirements. To this end, we have associated each pattern with its semantics in UPPAAL formalism that can formally capture it. From each requirement, elements of the automaton are derived by transforming all patterns which forms the requirement, one after the other in the order of their appearance. The details of the translation of all the patterns are presented in Table I. Note that in the description, the parts already generated and which do not change from one step to another are greyed. In particular, we see each state of a requirement is translated to a location with the same name, a condition to a guard, etc. Note that an event is generated and initiated each time a condition is built, not only when a trigger event takes place. Indeed, the requirements specification is based on an eager semantics: transitions are fired as soon as they are enabled. However, UPPAAL considers all transitions as lazy: when a transition is enabled, the system can choose to react immediately or to wait. To cope with this issue, we declare all generated synchronisation channels as urgent, to enforce that the transitions will be taken immediately when the condition is satisfied. Moreover, when the translation does not generate a synchronisation event, to ensure the immediate execution of the transitions we resort to a modelling artefact with the use of the special event denoted **now** emitting over a broadcast channel. At the end of this step, all state-driven requirements are addressed and an initial automata is then built.
2) During the second phase, the model is complemented by transitions and states representing the trigger events that may occur. In particular, we see an event-driven pattern is translated into a corresponding synchronisation channel, which completes a previously generated transition.
3) The model is also complemented in a third phase through the treatment of action-driven requirements. The side effect expressions related on certain transitions are

completed (using the ⊎ operator) to produce a complete update (complete actions to be executed when these transitions are fired). At the end of these three steps, a complete automata is built.

4) The fourth phase is the pruning phase that deals with constraint-type requirements. These requirements modify the structure of the resulting automata, by pruning the non-valid transitions. Indeed, as this kind of requirements explicitly express undesirable situations. Then, based on them, the model is modified and corrected to meet their contacts. After all constraints have been applied, the final automaton is built.
5) During the final phase, other automata of the global model are generated. They are derived from the last family of requirements, those that describe when certain events are issued. These requirements are in a limited number and they have generic templates. To each template we associated a generic model which is instantiated when a matching requirement is met.

Once the translation procedure is complete, if it is possible to generate a valid result (model), this constitutes evidence that the set of requirements is consistent, correct, and feasible. This initial model may be supplemented, if necessary, by additional information that will give the engineers reference points for missing requirements. Only the properties that cannot be enforced by design need for a posteriori verification.

The different processes of our framework are automatic, except the preprocessing phase that is performed in cooperation with the domain experts in charge of system design.

V. Case Study: Automatic Park Assist

To illustrate our approach, we use as example a feature available in almost all self-driving cars. The advanced driver-assistance systems (ADAS) are a key underlying technology in emerging autonomous vehicles [28]. They include several functions for automated driving, among them the Automatic Park Assist (APA) function that assists the drivers (see Fig.3) with parking safely and accurately. APA function provides easy parking by identifying sufficient parking spaces and steering the vehicle into it. The parking system can be supervised by the driver, who can override the operation pushing the accelerator pedal or the brake pedal. It can be fully automatic by an activation of the driver on the control broad, then APA function fully takes over control of parking functions, including steering, braking, shifting, and acceleration, to assist drivers in parking. To position the vehicle for parking, it gathers information from different types of sensors, such as ultrasonic sensors, lidar, camera, evaluate the situation. It subsequently sends control signal to actuators. When these latter receive control signal, active steering or braking subsystem execute instructions effectively and efficiently. Information is exchanged between components by updating the signals shared between them.

The specification of this function is defined in a STRComp including 400 textual requirements, but also requirements in the form of use-cases, tables and graphics. We carefully

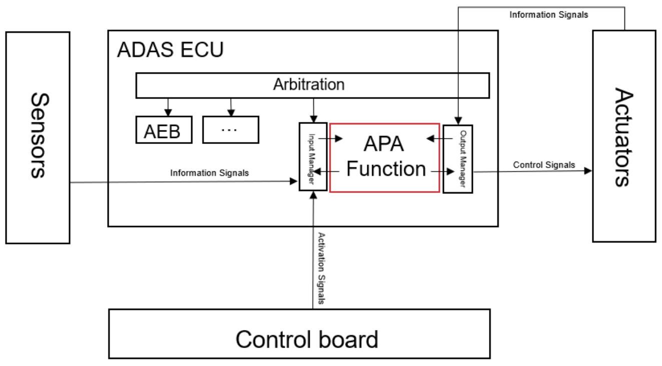


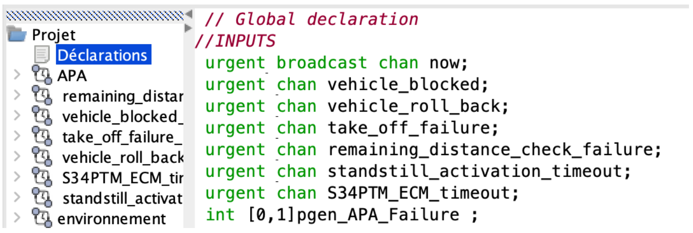
Fig. 3: Interactions between APA function and other components

studied the STRComp documents. They provide the overall description of the function, including details about interactions with other subsystems. Our study reveals that 70% of the requirements gleaned from the documents are interaction-specific while the remaining 30% are functional-specific. Their analysis helped in identifying 41 signals: 16 originate from the SONAR sensors, 17 from the Around View Monitor (AVM) system, and 18 move towards various actuators such as the steering and the braking. We will illustrate our explanation of the different phases through examples of requirements.

Declaration part: Let us consider the interface requirement `REQ_01` that defines the incoming signal `pgen_APA_Failure` and its possible values `vgen_NoFailure` and `vgen_Failure`.

```
REQ_01: APA system shall process and receive from SONAR
        the signal pgen_APA_Failure with the following values:
              - vgen_NoFailure
              - vgen_Failure
```

We define signals within UPPAAL tool as variables. For practical purposes, we encoded all symbolic values of signals by numerical values (int type). Indeed, UPPAAL does not support type String. Fig. 4 shows a small portion of the interface showing the declarations, we see the signal `pgen_APA_Failure` can have values 0 or 1 corresponding to `vgen_NoFailure` or `vgen_Failure` respectively.



```
// Global declaration
//INPUTS
urgent broadcast chan now;
urgent chan vehicle_blocked;
urgent chan vehicle_roll_back;
urgent chan take_off_failure;
urgent chan remaining_distance_check_failure;
urgent chan standstill_activation_timeout;
urgent chan S34PTM_ECM_timeout;
int [0,1]pgen_APA_Failure ;
```

Fig. 4: Screenshot of a part of the declaration

Automata construction: The analysis of all the functional requirements reveals that the function consists of six states (positions) `in_maneuver`, `out_maneuver`, `safe_state_1`, `safe_state_2`, `safe_state_3` and `safe_state_4`

TABLE I: Patterns and their associated model excerpts

	Requirement patterns	their associated semantics
State-driven #1	(a) $\boxed{\textbf{while } \langle state \rangle \textbf{ and } \langle condition \rangle}$	$guard = [\![condition]\!]$ $\langle state \rangle \xrightarrow{guard}$
	(b) $\boxed{\textbf{when } \langle trigger \rangle}$ - - - - - - - - - - -	$event = [\![trigger]\!]$ $\langle state \rangle \xrightarrow{guard\ event?}$ - - - - - - - - - - - - - - urgent broadcast chan now $\langle state \rangle \xrightarrow{guard\ now!}$
	(c) $\boxed{\langle system \rangle \textbf{ shall}}$ $\boxed{\textbf{switch to } \langle state \rangle}$ - - - - - - - - - - - $\boxed{\langle action \rangle}$	$\langle state \rangle \xrightarrow{guard\ event?} \langle state \rangle$ - - - - - - - - - - - - - - - - - - - $update = [\![action]\!]$ $\langle state \rangle \xrightarrow[update]{guard\ event?}$
Event-driven #2	$\boxed{\textbf{when } \langle trigger \rangle, \langle system \rangle \textbf{ shall switch to } \langle state \rangle}$	Let S be the set of all states generated in step #1 $event = [\![trigger]\!]$ For all $s \in S$, $s \xrightarrow{event?} \langle state \rangle$
Action-driven #3	$\boxed{\textbf{when entering } \langle state \rangle, \langle system \rangle \textbf{ shall } \langle action \rangle+}$	$update' = [\![action+]\!]$ for all transition $\xrightarrow[update]{guard\ event?} \langle state \rangle$ $\xrightarrow[update \uplus update']{guard\ event?} \langle state \rangle$

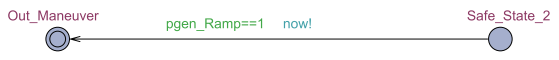
which are modelled in UPPAAL by six locations. These locations combined with the different signal values give several thousand of actual states.

To illustrate how the procedure operates let us apply the translation rules on a small subset of requirements.

1) Let us start with the following requirement:

```
REQ_02: while APA system is in Safe_State_2 and
        pgen_Ramp = vgen_on,
        APA system shall switch to Out_Maneuver
```

that is a state-driven requirement by applying the appropriate rule, the translation generates the following transition:

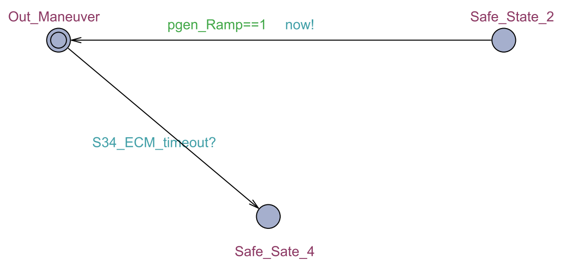


Observe the use of the urgent broadcast synchronisation channel **now** to enforce the immediate crossing of the transition when the condition is satisfied.

2) The translation of another requirement of the same type:

```
REQ_03: while Out_Maneuver, when S34_ECM_timeout,
        APA system shall switch to Safe_State_4
```

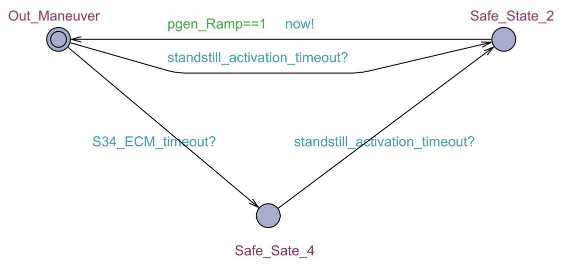
generates another transition from the state `Out_Maneuver` to `Safe_State_4`:



3) Then, let us consider the following event-driven requirement:

```
REQ_04: when standstill activation timeout,
        APA system shall switch to Safe_State_2
```

The translation of this requirement completes the model as follows:

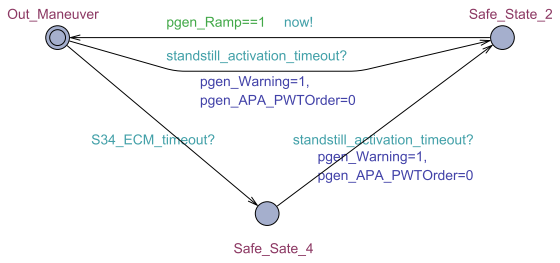


Observe that a transition going from all already generated states (`Out_Maneuver` and `Safe_State_4`) to the state `Safe_State_2` and labelled with a triggered event is generated.

4) Let us now consider an example of action-driven requirement:

```
REQ_05: when entering Safe_State_2  APA system shall:
        - set pgen_Warning to vgen_Alert
        - release powertrain control
```

its translation allows the completion of the action part as shown in the following figure:

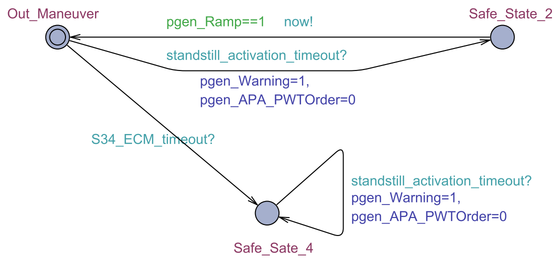


Observe that two actions are added to the two transitions entering the state `Safe_State_2`: – (`pgen_Warning`=1) is a direct translation of the first action; while the action – (`pgen_APA_PWTOrder`=0) results from the treatment of another requirement (which is not detailed here).

5) Once all requirements have been processed and the elements of the automaton created, the constraint requirements are applied. To show how they operate on the model, let us consider the following constraint:

```
REQ_06: if APA system is in Safe_State_4 and entrance
        conditions to Safe_State_2 are satisfied,
        APA system shall switch to Safe_State_4
```

The model is then transformed into the following:

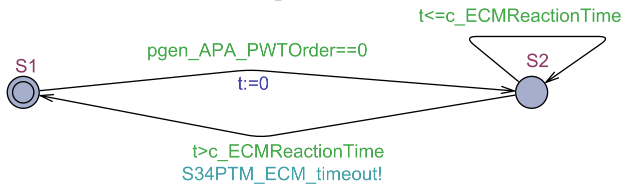


Observe that as specified by the constraint the transition going to state `Safe_State_2` is redirected to state `Safe_State_4`.

6) Finally, let us consider the following requirement that allows the triggering of the event `S34_ECM_timeout`:

```
REQ_07: APA system shall detect,
        if pgen_APA_PWTOrder = vgen_NotRequested
        for more than c_ECMReactionTime
```

the result of its translation is the following timed automaton which completes the main automaton:



The outcome of the translation of the APA system requirements is a collection of 8 automata: one main automaton which models the behaviour of the function, and additional automata modelling all the events that cause the evolution of the behaviour. The main automaton is given in Fig. 5. Although it appears to be small in size, the number of states obtained by product with the seven other automata (which are similar to the automaton corresponding to the requirement `REQ_07`) is around 37 thousand states.

Building a valid model offers an early assurance of the correctness and consistency of requirements. In addition to provide evidence that the system is realisable, the generated model from a set of requirements can be used for a posteriori verification. It may also be used as an aid to deep understanding in early phase requirements engineering. With this in mind, we used the generated model to check some properties.

First, we started by verifying (using UPPAAL model checker) usual properties such as the verification of deadlock freedom which is essential when combining concurrent components. This property is expressed in CTL language as:

- A[] NOT DEADLOCK

This property is evaluated to TRUE meaning that the model is deadlock-free.

Next, we proved a formula that checks the reachability of all system states. For example, the property:

- E <> APA.IN_MANEUVER

expressing that there exists at least one path starting from initial state along which state IN_MANEUVER will be reached. The property holds for this state and for all others, except for `Safe_State_3` and `Safe_State_4`. Not surprisingly, indeed these two states are reachable by firing condition dealing with signals updated by an external component, typically the SONAR. However, we did not include this component in our model, so the update will never be performed and therefore the condition will never be enabled.

Afterwards, we proved properties related to the system but not described in the requirements specification. For example, having the knowledge about the value of certain signals when the function is in a given state, we check whether this property is complied with. For instance, we know that the Flashing Indicator and the Braking signals have to be available only when the system is `In_maneuver` state. By using the corresponding signals in the model this can be expressed by safety properties as follows:

- A[] APA.OUT_OF_MANEUVER IMPLY PGEN_APA_FLASHINGINDICATORREQUEST_IN==0
- A[] APA.OUT_OF_MANEUVER IMPLY PGEN_APA_BRAKEWHEELTORQUEORDER_APAPARK==0

Both properties are evaluated to TRUE meaning that the specification as defined is compliant with expected behaviour, that is to say that the set of requirements covered describe the expected behaviour.

Discussion: This pilot study shows the potential of the proposed approach. Our approach is based on predefined set of templates, but it can accommodate additional templates for requirements specification, provided that they are associated with automata semantics. However, to be adopted in an industrial context some issues remain and need to be addressed. These include the effectiveness of the approach and the use of the analysis results, that might be addressed in future work. Generated models can be used to help discussion and to explore and learn about the engineer's needs. The information provided by UPPAAL model checker to the users for feedback, including animations, simulations, and derived counter-examples, assists in this regards. However, to apply remedial measures if needed, it is necessary that the negative feedback be viewed on textual representation of the system requirements.

The scalability issues in the context of industrial verification still need to be resolved. The approach presented in the paper pursues this objective by attempting to reduce the time and cost of the verification phase. The environment or context in which a system will run is often not taken into account. Reasoning about these aspects and their integration into the tool is among the point that requires further work. It is necessary to model not only the behaviour of the target system but also the environment interacting with the system. In UPPAAL, contrary to NuSMV model checker, the system variables cannot change via external interactions with the environment. So in order to simulate the system there is a need to model the environment. The typical approach to model the environment is to build an automaton which updates the values of all signals used by the system non-deterministically. Although it allows the exhaustive control of all possible execution sequences. It tends to generate a large number of instances of a single model and consequently leads to state-explosion. There is need to other approach that enables to filter out uninterested input values from all possible values.

The applicability of our approach naturally depends on the size of the generated models and is therefore limited by the capacity of the model checker used. In this work, we have used a symbolic model checker precisely to challenge this limitation. We are aware that there are solutions that can be used to alleviate the problem of state explosion, such as the adoption of a compositional analysis approach or the use of modular model checking algorithms.

VI. Related Works

Plenty research works for the requirements analysis have been presented in the scientific literature, most of them focus on new or improved techniques for evaluating the quality of requirements. They look for ill-formedness or errors in requirements, where an error can be inconsistency, incompleteness or ambiguity [8], [27], [30]. There have been very few tools that support such analysis on real-world applications and in an industrial setting. The lack in tooling is partly due to the use of natural language. Indeed, natural language is the dominant form of expression of requirements in practical projects in industry. Such approaches face a fundamental challenge: writing requirements and designing system requires a high degree of precision and accuracy, but natural language is inherently imprecise.

Significant efforts, including both research efforts and industrial products development, have been made to improve the techniques for analysing the quality of requirements. Some approaches use partially formalized notations or semi-formal languages such as Doors [2], Reqtify [1], and SysML [16]. Doors and Reqtify are widely used in industry, they benefit from mature tools, e.g. [4] and [3] developed by major companies, IBM Rational and Dassault Systems respectively. In both cases the focus is not on analysing requirements and verification methodology, but on managing requirements.

Another approach introduces the notion of Constrained Natural-Languages (CNL) such as EARS [24], RELAX [31] and Stimulus [20] languages. These languages with a simplified syntax and restricted lexical terms help to bridge the gap between informal and formal representation of requirements, and improve their translation from informal to formal. Such research is often accompanied by proofs-of-concepts or pilot studies that show the potential of the proposed ideas. Some research, such as the approach described in the paper [17] is concerned with the translation of requirements into properties that can be used with finite-state verification tools, while our approach aims at building state machines for verification of functional properties. To the best of our knowledge, only Stimulus is supported by a tool [21], both the language and the tool have the same name. It enables engineers to generate test vectors and test objectives automatically, that can be used to check whether the developed system is compliant with its specification. Our approach is similar to to the stimulus approach, in the sense we generate state machines from CNL templates. An important difference from this approach, we aim at the transformation of system requirements into a model, that provides early evidence that the requirement specifications are realizable, as opposed to the test objectives that the system should meet.

Formal methods have proven their cost-effectiveness through their successful use in industrial context and in different areas, such as railway [13], aeronautics and aerospace [9]. There exist a number of approaches relying on mathematical theories of graphs and automata for requirements analysis. Such approaches use graphical notations, such as infinite automata, state diagrams and statecharts, to specify requirements. However, these representations make their use less practical on real-world applications, in particular in an industrial setting. Requirements can use other mathematical theories for system modelling and analysis, for example Event-B [5] which is equipped with its own methodology based on set theory and predicate logic for modelling and to formally prove system correctness. This kind of mathematically rigorous tools, while they are powerful, are intended to specialists and engineers with deep understanding of their mathematical foundations and, also, applicable domains and limits.

In recent years the early validation of requirements and the use formal methods has become a focus for industrial research such as as mentioned in [10], [25] and [29]. Such empirical research needs to be supported by tools that can be used on real-world applications and in an industrial setting. In particular, the automotive industry has expressed interest in using such approach in developing complex automotive systems.

VII. Conclusion

This paper introduced a systematic process for building models from automotive requirements written in natural language with the aim to reduce the effort of testing and detects fixes late in software development lifecycle. The process is automated through a tool and existing verification tool.

The first effort led to categorize the entire set of requirements of the system under design in order to formalize them. We provided a proof of concept regarding our approach by designing models and verifying some properties. We verified general properties using UPPAAL such as liveness and reachability. Naturally our next goal is to find some new type of properties to be verified in order to validate safety critical aspect and also detect flaws. Another goal can be to propose a requirement specific language for the expression of requirement in the automotive domain to match standard such as AUTOSAR [18]. In [32], we have given some directions as well limited to a smaller set of requirements. This language is different from the languages that already exist for the description of automotive standards, as [15] and [11]. In the sense, it allows the specification of systems at a high abstraction level, without any prior knowledge about architectural considerations and how the functions are then allocated to the components of the physical architecture.

References

[1] Dassault system–reqtify. https://www.3ds.com/fr/produits-et-services/catia/produits/reqtify/.

[2] Ibm–rational doors. http://www-03.ibm.com/software/products/ratidoor.

[3] The reuse company–rat. https://www.reusecompany.com/rat-authoring-tools.

[4] The reuse company–rqa. https://www.reusecompany.com/rqa-quality-studio.

[5] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, USA, 1st edition, 2010.

[6] AFNOR. National Survey: The costs of poor quality in industry. Technical report, AFNOR, October 2017.

[7] Thomas Baar. Verification Support for a State-Transition-DSL Defined with Xtext. In *Lecture Notes in Computer Science, vol 9609*, pages 50–60. Springer, 06 2016.

[8] Daniel M. Berry and Erik Kamsties. *Ambiguity in Requirements Specification*, pages 7–44. Springer US, Boston, MA, 2004.

[9] Jean-Louis Boulanger. *Industrial Use of Formal Methods: Formal Verification*. ISTE Ltd. Wiley, 2013.

[10] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Panagiotis Katsaros, Konstantinos Mokos, Viet Yen Nguyen, Thomas Noll, Bart Postma, and Marco Roveri. Spacecraft early design validation using formal methods. *Reliability Engineering & System Safety*, 132:20–35, 2014.

[11] Stefan Bunzel. AUTOSAR the Standardized Software Architecture. *Informatik-Spektrum*, 34:79–83, 2011.

[12] Robert N Charette. *Software engineering environments : concepts and technology*. Intertext Publications, New York, NY, 1986.

[13] X. Chen, Z. Zhong, Z. Jin, M. Zhang, T. Li, X. Chen, and T. Zhou. Automating consistency verification of safety requirements for railway interlocking systems. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pages 308–318, 2019.

[14] B. H. C. Cheng and J. M. Atlee. Research directions in requirements engineering. In *Future of Software Engineering (FOSE '07)*, pages 285–303, 2007.

[15] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. *The EAST-ADL Architecture Description Language for Automotive Embedded Software*, pages 297–307. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[16] Michel dos Santos Soares and Jos L. M. Vrancken. Requirements specification and modeling through sysml. *2007 IEEE International Conference on Systems, Man and Cybernetics*, pages 1735–1740, 2007.

[17] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, pages 411–420. ACM, 1999.

[18] Simon Fürst and Markus Bechter. AUTOSAR for connected and autonomous vehicles: The AUTOSAR adaptive platform. In *DSN Workshops*, pages 215–217. IEEE Computer Society, 2016.

[19] IEEE. Systems and software engineering – Life cycle processes – Requirements engineering. *ISO/IEC/IEEE 29148:2011(E).*

[20] Bertrand Jeannet and Fabien Gaucher. Debugging Real-Time Systems Requirements: Simulate The "What" Before The "How". In *Embedded World Conference, Nürnberg, Germany*, 2015.

[21] Bertrand Jeannet and Fabien Gaucher. Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse,France, 2016.

[22] Herb Krasner. The Cost of Poor Quality Software in the US: A 2018 Report. Technical report, CISQ Consortium for IT Software Quality, September 2018.

[23] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, December 1997.

[24] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. Easy Approach to Requirements Syntax (EARS). In *2009 17th IEEE International Requirements Engineering Conference*, pages 317–322, 2009.

[25] Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *International Journal on Software Tools for Technology Transfer*, 8(4):303–319, 2006.

[26] Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats Per Erik Heimdahl. Early validation of requirements through formal methods. *International Journal on Software Tools for Technology Transfer*, 8(4-5):303–319, 2006.

[27] Paul Rayson, Ieee Computer Society, Ken Cosh, and Ieee Computer Society. K.: Shallow Knowledge as an Aid to Deep Understanding in Early Phase Requirements Engineering. *IEEE Transactions on Software Engineering*, pages 969–981, 2005.

[28] Y. Song and C. Liao. Analysis and review of state-of-the-art automatic parking assist system. In *2016 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, pages 1–6, 2016.

[29] Emmanouela Stachtiari, Anastasia Mavridou, Panagiotis Katsaros, Simon Bliudze, and Joseph Sifakis. Early validation of system requirements and design through correctness-by-construction. *Journal of Systems and Software*, 145:52–78, 2018.

[30] Kimberly S. Wasson. A Case Study in Systematic Improvement of Language for Requirements. In *Proceedings of the 14th International Requirements Engineering Conference*, pages 6–15. IEEE Computer Society, 2006.

[31] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. *2009 17th IEEE International Requirements Engineering Conference*, pages 79–88, 2009.

[32] Assioua Yasmine, Ameur-Boulifa Rabea, and Guitton-Ouhamou Patricia. Towards Formal Verification of Autonomous Driving Supervisor Functions. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, January 2020.

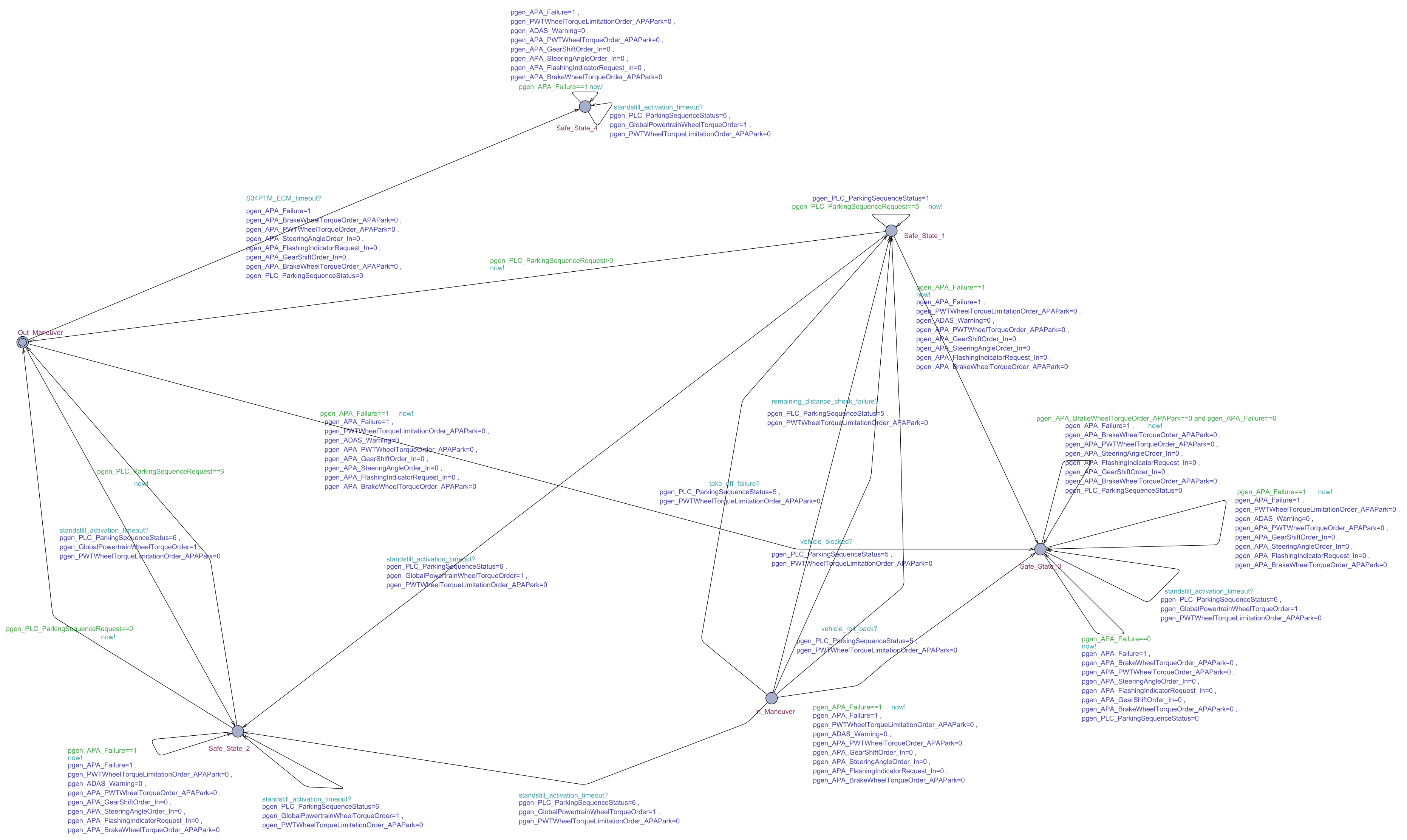

Out_Maneuver
In_Maneuver
Safe_State_1
Safe_State_2
Safe_State_3
Safe_State_4
pgen_APA_Failure=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0 , pgen_ADAS_Warning=0 , pgen_APA_PWTWheelTorqueOrder_APAPark=0 , pgen_APA_GearShiftOrder_In=0 , pgen_APA_SteeringAngleOrder_In=0 , pgen_APA_FlashingIndicatorRequest_In=0 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0
pgen_APA_Failure==1 now!
standstill_activation_timeout? pgen_PLC_ParkingSequenceStatus=6 , pgen_GlobalPowertrainWheelTorqueOrder=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0
S34PTM_ECM_timeout? pgen_APA_Failure=1 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0 , pgen_APA_PWTWheelTorqueOrder_APAPark=0 , pgen_APA_SteeringAngleOrder_In=0 , pgen_APA_FlashingIndicatorRequest_In=0 , pgen_APA_GearShiftOrder_In=0 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0 , pgen_PLC_ParkingSequenceStatus=0
pgen_PLC_ParkingSequenceStatus=1
pgen_PLC_ParkingSequenceRequest==5 now!
pgen_PLC_ParkingSequenceRequest&gt;0 now!
pgen_APA_Failure==1 now! pgen_APA_Failure=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0 , pgen_ADAS_Warning=0 , pgen_APA_PWTWheelTorqueOrder_APAPark=0 , pgen_APA_GearShiftOrder_In=0 , pgen_APA_SteeringAngleOrder_In=0 , pgen_APA_FlashingIndicatorRequest_In=0 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0
remaining_distance_check_failure? pgen_PLC_ParkingSequenceStatus=5 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0
pgen_APA_BrakeWheelTorqueOrder_APAPark==0 and pgen_APA_Failure==0
pgen_APA_Failure=1 , now! pgen_APA_BrakeWheelTorqueOrder_APAPark=0 , pgen_APA_PWTWheelTorqueOrder_APAPark=0 , pgen_APA_SteeringAngleOrder_In=0 , pgen_APA_FlashingIndicatorRequest_In=0 , pgen_APA_GearShiftOrder_In=0 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0 , pgen_PLC_ParkingSequenceStatus=0
pgen_APA_Failure==1 now! pgen_APA_Failure=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0 , pgen_ADAS_Warning=0 , pgen_APA_PWTWheelTorqueOrder_APAPark=0 , pgen_APA_GearShiftOrder_In=0 , pgen_APA_SteeringAngleOrder_In=0 , pgen_APA_FlashingIndicatorRequest_In=0 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0
pgen_APA_Failure==1 now! pgen_APA_Failure=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0 , pgen_ADAS_Warning=0 , pgen_APA_PWTWheelTorqueOrder_APAPark=0 , pgen_APA_GearShiftOrder_In=0 , pgen_APA_SteeringAngleOrder_In=0 , pgen_APA_FlashingIndicatorRequest_In=0 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0
pgen_PLC_ParkingSequenceRequest==6 now!
take_off_failure? pgen_PLC_ParkingSequenceStatus=5 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0
vehicle_blocked? pgen_PLC_ParkingSequenceStatus=5 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0
standstill_activation_timeout? pgen_PLC_ParkingSequenceStatus=6 , pgen_GlobalPowertrainWheelTorqueOrder=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0
standstill_activation_timeout? pgen_PLC_ParkingSequenceStatus=6 , pgen_GlobalPowertrainWheelTorqueOrder=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0
standstill_activation_timeout? pgen_PLC_ParkingSequenceStatus=6 , pgen_GlobalPowertrainWheelTorqueOrder=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0
pgen_PLC_ParkingSequenceRequest==0 now!
vehicle_roll_back? pgen_PLC_ParkingSequenceStatus=5 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0
pgen_APA_Failure==0 now! pgen_APA_Failure=1 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0 , pgen_APA_PWTWheelTorqueOrder_APAPark=0 , pgen_APA_SteeringAngleOrder_In=0 , pgen_APA_FlashingIndicatorRequest_In=0 , pgen_APA_GearShiftOrder_In=0 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0 , pgen_PLC_ParkingSequenceStatus=0
pgen_APA_Failure==1 now! pgen_APA_Failure=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0 , pgen_ADAS_Warning=0 , pgen_APA_PWTWheelTorqueOrder_APAPark=0 , pgen_APA_GearShiftOrder_In=0 , pgen_APA_SteeringAngleOrder_In=0 , pgen_APA_FlashingIndicatorRequest_In=0 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0
standstill_activation_timeout? pgen_PLC_ParkingSequenceStatus=6 , pgen_GlobalPowertrainWheelTorqueOrder=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0
pgen_APA_Failure==1 now! pgen_APA_Failure=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0 , pgen_ADAS_Warning=0 , pgen_APA_PWTWheelTorqueOrder_APAPark=0 , pgen_APA_GearShiftOrder_In=0 , pgen_APA_SteeringAngleOrder_In=0 , pgen_APA_FlashingIndicatorRequest_In=0 , pgen_APA_BrakeWheelTorqueOrder_APAPark=0
standstill_activation_timeout? pgen_PLC_ParkingSequenceStatus=6 , pgen_GlobalPowertrainWheelTorqueOrder=1 , pgen_PWTWheelTorqueLimitationOrder_APAPark=0

Fig. 5: Main automata for Automatic Park Assist function